

# 浙江大学

## 本科实验报告

课程名称：计算机体系结构

姓 名：范源颢

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：计算机科学与技术

学 号：3180103574

指导教师：陈文智

2020 年 30 月 30 日

# 浙江大学实验报告

课程名称： 计算机体系结构 实验类型： 综合

实验项目名称： 五阶段 CPU 的流水线执行/有停顿的流水线 CPU

学生姓名： 范源颢 专业： 计算机科学技术 学号： 3180103574

同组学生姓名： 周寒靖 指导老师： 陈文智

实验地点： 曹光彪西楼-301 实验日期： 2020 年 10 月 30 日

## 一、 实验目的和要求

目的：

1. 了解 CPU 流水线原理
2. 了解流水线 CPU 的基本单元；
3. 了解 5 级工作流程；
4. 了解流水线 CPU 停顿原理
5. 了解数据冒险（data Hazard）原理
6. 掌握流水线 CPU 停顿检测和停顿流水线的方法。
7. 掌握 CPU 核心模块的仿真
8. 掌握有停顿流水线 CPU 的程序验证方法。

要求：


1. 设计 CPU 控制器
2. 根据 CPU 控制器，设计一个 5 阶段流水线 CPU 的数据通路，包含：
  - 5 阶段寄存器；
  - 寄存器文件；
  - 内存（指令内存和数据内存）
  - 其他基本单元；
3. 设计 5 阶段流水线 CPU 数据通路的停顿处理部分

4. 增加停顿条件检测
5. 对 CPU 核进行仿真
6. 利用程序验证 CPU 功能并且观察程序的执行。

## 二、 实验内容和原理

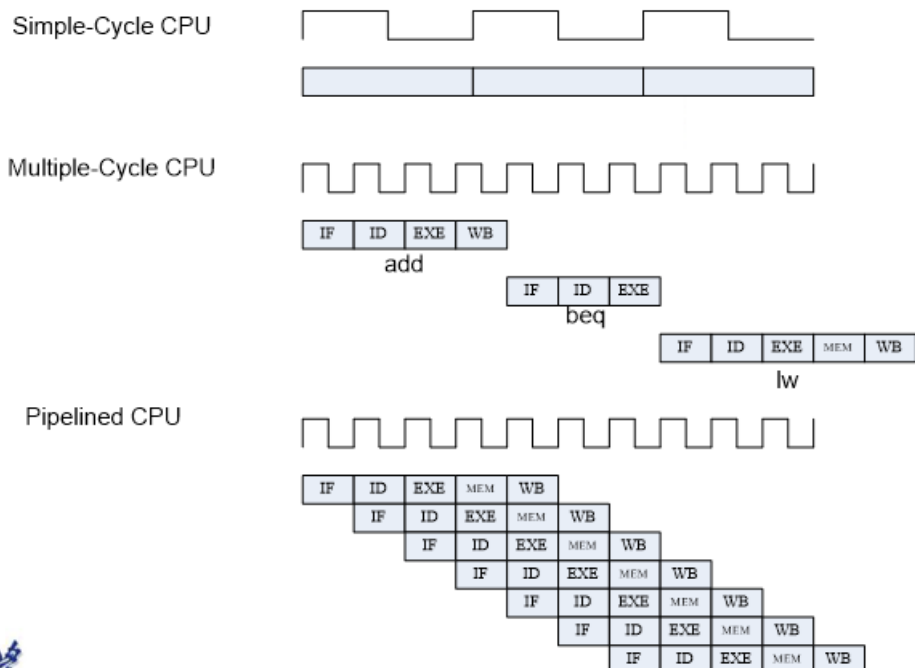
2.1 我们在本次试验中依旧实现 16 指令的 MIPS 汇编语言。

各条语句的语法和含义如下：



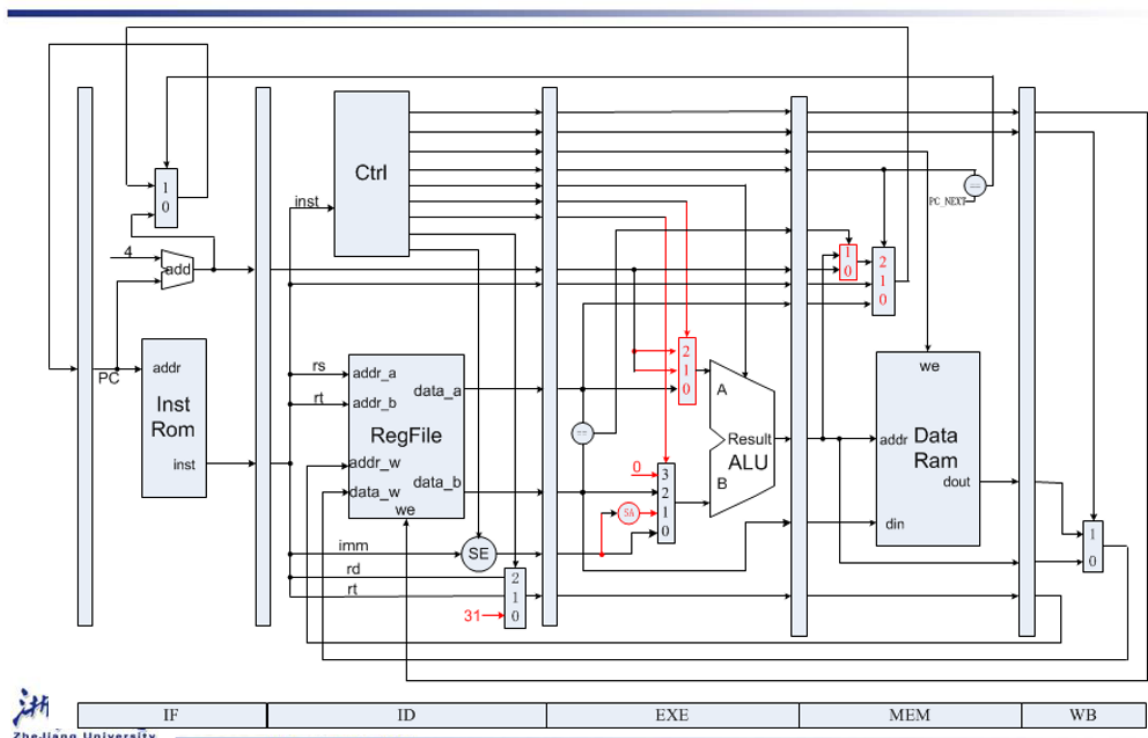
16 MIPS Instructions							Operations
Bit #	31..26	25..21	20..16	15..11	10..6	5..0	
R-type	op	rs	rt	rd	sa	func	
<u>add</u>		rs	rt	rd	00000	100000	$rd = rs + rt$ ; with overflow PC+=4
<u>sub</u>		rs	rt	rd	00000	100010	$rd = rs - rt$ ; with overflow PC+=4
<u>and</u>		rs	rt	rd	00000	100100	$rd = rs \& rt$ ; PC+=4
<u>or</u>		rs	rt	rd	00000	100101	$rd = rs   rt$ ; PC+=4
<u>sll</u>	000000	00000	rt	rd	sa	000000	$rd = rt \ll sa$ ; PC+=4
<u>srl</u>		00000	rt	rd	sa	000010	$rd = rt \gg sa$ (logical); PC+=4
<u>slt</u>		rs	rt	rd	00000	101010	if( $rs < rt$ ) $rd = 1$ ; else $rd = 0$ ; <(signed) PC+=4
<u>jr</u>		rs	00000	00000	00000	001000	$PC = rs$
I-type	op	rs	rt	immediate			
<u>addi</u>	001000	rs	rt	imm			$rt = rs + (\text{sign\_extend})imm$ ; with overflow PC+=4
<u>andi</u>	001100	rs	rt	imm			$rt = rs \& (\text{zero\_extend})imm$ ; PC+=4
<u>ori</u>	001101	rs	rt	imm			$rt = rs   (\text{zero\_extend})imm$ ; PC+=4
<u>lw</u>	100011	rs	rt	imm			$rt = \text{memory}[rs + (\text{sign\_extend})imm]$ ; PC+=4
<u>sw</u>	101011	rs	rt	imm			$\text{memory}[rs + (\text{sign\_extend})imm] \leftarrow rt$ ; PC+=4
<u>beq</u>	000100	rs	rt	imm			if ( $rs == rt$ ) $PC += 4 + (\text{sign\_extend})imm \ll 2$ ; $PC += 4$
J-type	op	address					
<u>j</u>	000010	address					$PC = (PC+4)[31..28], \text{address} \ll 2$
<u>jal</u>	000011	address					$PC = (PC+4)[31..28], \text{address} \ll 2$ ; $\$31 = PC+4$

2.2 我们通过下图来表达流水线和单周期，多周期 CPU 的区别：



5 阶段流水线将每个程序分割成顺序执行 5 各阶段（IF 【Instruction Fetch，取回指令】，ID 【Instruction Decode，指令解码】， EXE 【EXEcution，执行运算】，MEM 【MEMory，读写寄存器】，WB 【WriteBack，写回寄存器】）流水线的要义在于，执行某个指令的某个阶段时，下一条指令中的前一个阶段可以同时执行，前提是被提前执行的指令不和当下正在执行的指令有冲突，否则需要引入停顿（stall）

### 2.3 5 阶段流水线 CPU 的逻辑电路图：



下方的阶段提示了执行相应阶段需要的硬件，竖条形的寄存器可以用“阶段 1/阶段 2” (例如 IF/ID)来命名。可以用于储存在一个阶段执行完毕下一个阶段开始之前的中间状态信息。

### 2.4 CPU 冒险（结构冒险，数据冒险和结构冒险）：

结构冒险：来源于资源冲突，电路的资源无法同时完成重叠的指令要求

数据冒险：来源于某个指令依赖于前一个指令的结果，于是在前一个指令执行完之前，后一个指令应该有所停顿。

结构冒险：来源于分支指令和其他类似的指令对于 PC 资源占有的冲突。

### 2.5 CPU 停顿的条件总结：

本次试验中我们不使用 fwd 算法,也即是,一旦出现冲突立即通过 stall(停顿)指令停顿整个流水线,使得后面的指令整体后移一个阶段。

PPT 中已经帮助总结了会引致停顿的两类情况,第一类是数据类型的冲突,第二类是 PC 寄存器跳转的冲突。

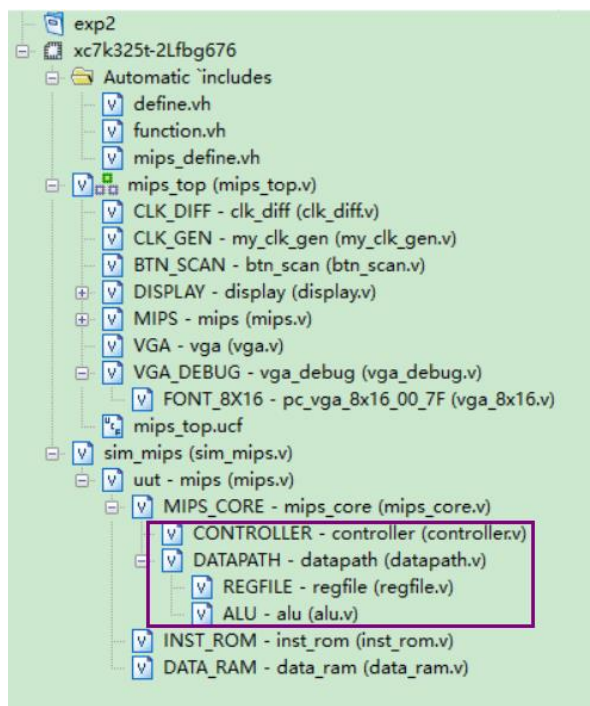
当一条指令位于 ID 阶段时,他之前的指令可能处于 EXE/MEM/WB 这三种阶段,如果处于 WB 阶段,则我们不需要考虑冲突,因为我们已经通过始终上下沿的分配使得写入寄存器必定先于读取寄存器,比较复杂的是当上一条指令处在 EXE 阶段或者 MEM 阶段而且 EXE 和 MEM 修改的寄存器正好是本条指令 ID 阶段需要读取的寄存器时,此时会有冲突发生(因为上一条指令的结果还没有写入到寄存器中,本条指令读取的值将是错误的)本次试验中,一旦这样的冲突发生,那么我们选择停顿流水线,使得之后的指令一律后退一个阶段。

这样的停顿条件可以表述为: `id.instr.rs == exe_instr.rd && exe_instr.writereg`,同理有 `id_instr.rt==exe_instr.rd && exe_instr.writereg`, `id_instr.rs==exe_instr.rd && mem_instr.writereg`, `id_instr.rt==exe_instr.rd && mem_instr.writereg`.

第二类情况就是当执行跳转指令时,我们要停止之后指令的执行,因为此时我们已经无法确定下一条指令是否是 PC+4 所读取的指令了,于是我们需要暂停流水线直到这些跳转指令执行完毕。具体来说,暂停流水线的情况包括:1.给 BEQ、Jump、Jal、Jr 等跳转指令解码的时候(利用 `pc_src != PC_NEXT` 即可);2.BEQ 指令在 EXE 阶段时(用 `is_branch_exe` 表示);2.BEQ 指令在读取内存的阶段(用 `is_branch_mem` 表示)。

### 三、 实验过程和数据记录

实验的文件结构如下:



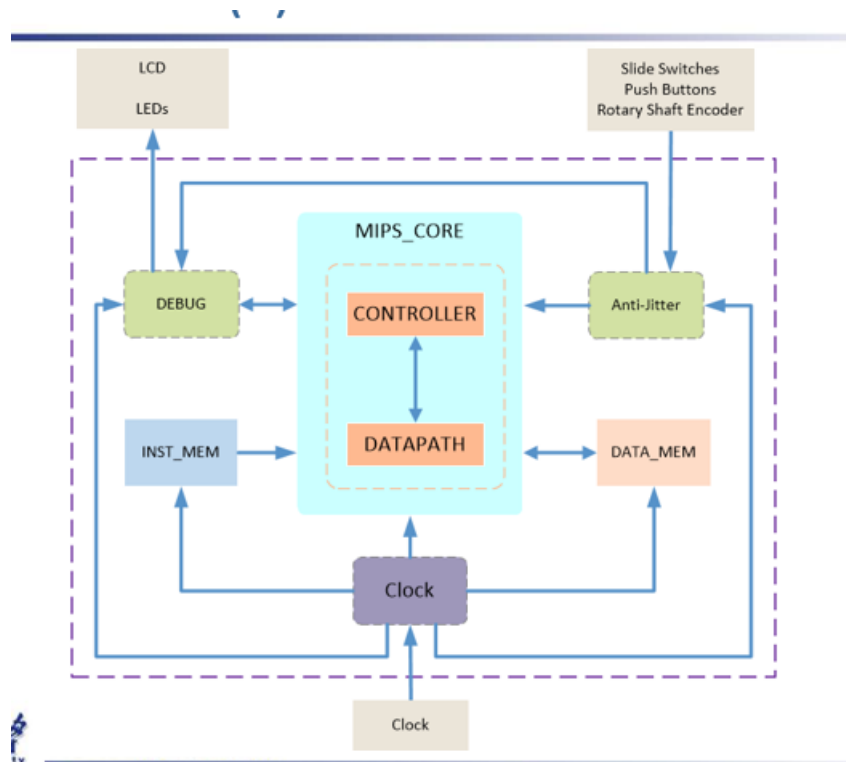
Automatic includes 文件夹下定义了我们可以在代码中使用的头文件；

mips\_top.v 是实验的主要工程，实现了对于 MIPS 指令的模拟，clk\_diff 和 clk\_gen 模块用于 CPU 内置时钟的信号生成的处理，便于各个进程的同步。Btn\_san 是处理输入信号的模块，时限完成的 mips\_top 模块将烧录到实验室的 sword 开发板上，通过按钮的控制反映 CPU 的计算情况，信息通过 3 个模块输出，分别是 DISPLAY 模块和 VGA，VGA\_DEBUG 模块，mips\_top.ucf 规定了电路的引脚，使得烧录之后的程序可以嵌入开发板；

sim\_mi.v 实际上是一个测试用的工程文件，用于在开发者的电脑上，用 Xilinx ISE 仿真烧录之后的输入输出过程，uut 文件指定了我们的模拟方式（通过内置时钟不断读取 inst\_rom 和 data\_ram 中的信息，并且把相关的信号反馈到电路图上），仿真检测的对象自然是 mips\_core，这个 mips\_core 核上文中 mips\_top 模块中的 mips\_core 是同一个文件。

我们需要完成 mips\_core 中的代码补全。mips.v 文件划分为 MIPS\_CORE 和 inst\_rom，data\_ram，完成了二者（CPU 和内存）的交互，mips\_core.v 划分为 controller.v 和 datapath.v，完成了二者（控制器和数据通路）的交互，我们重点需要完成 controller.v 和 datapath.v 的代码补全。

实验的逻辑模块如下：



本实验新增了很多控制信号，包括如下内容：

inst	Input for id instruction
is_branch_exe	pc src in exe is PC_NEXT or not
regw_addr_exe	address source to write data back to registers
wb_wen_exe	memory write enable signal
is_branch_mem	pc src in mem is PC_NEXT or not
regw_addr_mem	address source to write data back to registers
wb_wen_mem	memory write enable signal
regw_addr_wb	data source of data being written back to registers
wb_wen_wb	register write enable signal

pc_src	pc src: PC_NEXT, PC_JUMP, PC_JR, PC_BEQ
mem_ren	Mem read signal
mem_wen	Mem write signal
wb_addr_src	Control signal of write back addr
wb_data_src	Control signal of write back data
wb_wen	Control signal of write back enable
if_rst/if_en	Output for if stage controll
if_valid	Input for if stage stage
id_rst/id_en	Output for id stage controll
id_valid	Input for id stage stage
exe_rst/exe_en	Output for exe stage controll
exe_valid	Input for exe stage stage
mem_rst/mem_en	Output for mem stage controll
mem_valid	Input for mem stage stage
wb_rst/wb_en	Output for wb stage controll
wb_valid	Input for wb stage stage

这些信号的命名规则如下：

- 阶段名 if(instruction fetch),  
id(instruction\_decode),  
exe(execution),



mem(memory),

wb(write back)

- 信号名: `xxx_`: 在 `xxx` 阶段工作的信号

`_xxx`: 由 `xxx` 阶段产生的信号

例如: `wb_wen_mem`: MEM 阶段产生使得 writeback 阶段内存可写的信号。

## 1. CPU Datapath:

相比于 controller, datapath 其实需要改动的不大, 他所需要的一个原则是, 在每个时钟上沿检查 controller 指令的时候, 首先检查这个阶段的使能信号 (`if_en`, `id_en`, 等等) 是否被置位, 仅当这些信号为 1 的时候, datapath 才可以执行相应的工作, datapath 执行工作是通过不断传递 `inst_data` 和 `inst_addr` 实现的, 例如, 输入的 `inst_data` 如果在 `id` 阶段使能, 则他会被赋值给 `inst_data_id`, 之后我们从中取出 `addr_rs` 和 `addr_rd` 用于运算, 假如没有 `id` 阶段的使能信号 (`id_en`), 那么 `inst_data` 将不会赋值给 `inst_data_id`, 也不会有之后的执行。(然而, 这一层逻辑事实上在模板中已经实现好了, 我们的 datapath 需要补充的内容几乎和上一次作业一模一样)

我们依旧主要需要实现四个 CPU 控制信号对于数据通路的指示, 只是此时我们需要将上一次实验中的 `inst_addr` 等信号加上后缀, 明确是哪个阶段的信号:

Id 阶段产生的 wb 信号:

```
always @(*) begin
    regw_addr_id = inst_data_id[15:11];
    case (wb_addr_src_ctrl)
        WB_ADDR_RD: regw_addr_id = addr_rd; //
        WB_ADDR_RT: regw_addr_id = addr_rt; //
        WB_ADDR_LINK: regw_addr_id = GPR_RA; //
    endcase
end
```

EXE 阶段产生的信号:

```
always @(*) begin
    opa_exe = data_rs_exe;
    opb_exe = data_rt_exe;
```

```

    case (exe_a_src_exe)
        EXE_A_RS: opa_exe = data_rs_exe; //
        EXE_A_LINK: opa_exe = inst_addr_next_exe; //
        EXE_A_BRANCH: opa_exe = inst_addr_next_exe; //
    endcase
    case (exe_b_src_exe)
        EXE_B_RT: opb_exe = data_rt_exe; //
        EXE_B_IMM: opb_exe = data_imm_exe; //
        EXE_B_LINK: opb_exe = 32'h0; // linked address is the next one
of current instruction
        EXE_B_BRANCH: opb_exe = {data_imm_exe[29:0], 2'b0}; //
    endcase
end

```

其中，data\_rs\_exe，如前所述，是当 exe\_en 置位的时候，依据 data\_rs 赋值而来的值，data\_rs\_exe 是 EXE 阶段产生的信号，而 datapath 指挥 ALU 等操作单元的时候也是这个信号，所以其实这里应该命名为 exe\_data\_rs\_exe 更好，也许更加严格的命名规范可以使得代码的可读性进一步提升（但是模板中的代码还是不推荐修改）。

MEM 阶段（我们处理的是地址跳转的情况）

```

always @(*) begin
    case (pc_src_mem)
        PC_JUMP: branch_target_mem <=
{inst_addr_mem[31:28],inst_data_mem[25:0],2'b0}; //
        PC_JR: branch_target_mem <= data_rs_mem; //
        PC_BNE: branch_target_mem <=
rs_rt_equal_mem?inst_addr_next_mem:alu_out_mem; //
        PC_BEQ: branch_target_mem <=
rs_rt_equal_mem?alu_out_mem:inst_addr_next_mem; //
        default: branch_target_mem <= inst_addr_next_mem; // will
never used
    endcase
end

```

data\_rs\_mem 同理是 mem\_en 时，由 data\_rs\_exe 赋值而来的（依旧觉得这里命名为 mem\_data\_rs\_mem 更好），这里的 branch\_target\_mem 将会作为 if 阶段的输入，如下：

```

always @(posedge clk) begin
    if (if_rst) begin
        inst_addr <= 0;
    end
    else if (if_en) begin
        if (is_branch_mem)
            inst_addr <= branch_target_mem;
    end
end

```

```

        else
            inst_addr <= inst_addr_next;
        end
    end
end

```

最后，regw\_data\_wb 的写法依旧和上一次实验一致，注意加上后缀即可：

```

always @(*) begin
    regw_data_wb = alu_out_wb;
    case (wb_data_src_wb)
        WB_DATA_ALU: regw_data_wb = alu_out_wb; //
        WB_DATA_MEM: regw_data_wb = mem_din_wb; //
    endcase
end

```

至此完成 datapath 的填空。

## 2. CPU Controller

当我们得到 32 位的机器码时，我们需要根据机器码各个段位的内容将相应的内容抽取出来，同时根据机器码的语义发出相应的信号，具体和上一次实验的内容完全一样，唯一需要注意的是增加的 rs\_used/rt\_used 信号，表示这条指令是否需要用到相应的寄存器(但我们不考虑是否需要用到 rd,因为 waw 是不会造成冲突的)，比如：

```

case (inst[31:26])
    INST_R: begin
        case (inst[5:0])
            R_FUNC_JR: begin
                pc_src = PC_JR;
            end
            R_FUNC_ADD: begin
                ...;
            end
            R_FUNC_SUB: begin
                exe_alu_oper = EXE_ALU_SUB; //
                wb_addr_src = WB_ADDR_RD; //
                wb_data_src = WB_DATA_ALU; //
                wb_wen = 1;
                rs_used = 1; // // //
                rt_used = 1; // // //
            end
            R_FUNC_AND: begin
                ...;
            end
            R_FUNC_OR: begin
                ...;
            end
        end
    end
end

```

```

        end
        R_FUNC_SLT: begin
            ...;
        end
        default: begin
            unrecognized = 1;
        end
    endcase
end
...;
endcase

```

其他指令的解码我们不再赘述，直接看流水线的控制：

```

// pipeline control
reg reg_stall;
reg branch_stall;
wire [4:0] addr_rs, addr_rt; ///这一条要执行的语句的地址

assign
    addr_rs = inst[25:21],
    addr_rt = inst[20:16];

always @(*) begin ///PPT27 页
    reg_stall = 0;
    if (rs_used && addr_rs != 0) begin ///ID instr.rs = exe instr.rd
and exe instr.writereg, 本条 rs 和下一条 WB 阶段的 rd 冲突；此时不考虑 0 号寄存器，
因为他一直是 0 不会冲突
        if (regw_addr_exe == addr_rs && wb_wen_exe) begin
            reg_stall = 1;
        end
        else if (regw_addr_mem == addr_rs && wb_wen_mem) begin ///id
instr.rs=mem instr.rd and mem instr.writereg 本条 rs 和下一条 MEM 阶段的 rd
冲突；同样不考虑 0 号
            reg_stall = 1;
        end
    end
    if (rt_used && addr_rt != 0) begin
        if (regw_addr_exe == addr_rt && wb_wen_exe) begin ///ID
instr.rt = exe instr.rd and exe instr.writereg, 本条 rt 和下一条 WB 阶段的 rd
冲突；同样不考虑 0 号
            reg_stall = 1;
        end
        else if (regw_addr_mem == addr_rt && wb_wen_mem) begin ///:
id instr.rt=mem instr.rd and mem instr = instr.writereg, 本条 rt 和下一条
MEM 阶段的 rd 冲突；同样不考虑 0 号
    end
end

```

```

        reg_stall = 1;
    end
end
end

always @(*) begin///PPT28
    branch_stall = 0;
    if (pc_src != PC_NEXT || is_branch_mem || is_branch_exe)///
        branch_stall = 1;
end

```

上文的代码只是将本文 2.5 节的内容直译了一下，值得注意的是 branch\_stall 和 reg\_stall 的用法，在模板中，我们看到：

```

always @(*) begin
    ...;
    // this stall indicate that ID is waiting for previous instruction,
    should insert NOPs between ID and EXE.
    else if (reg_stall) begin
        if_en = 0;
        id_en = 0;
        exe_rst = 1;
    end
    // this stall indicate that a jump/branch instruction is running,
    so that 3 NOP should be inserted between IF and ID
    else if (branch_stall) begin
        id_rst = 1;
    end
end
end

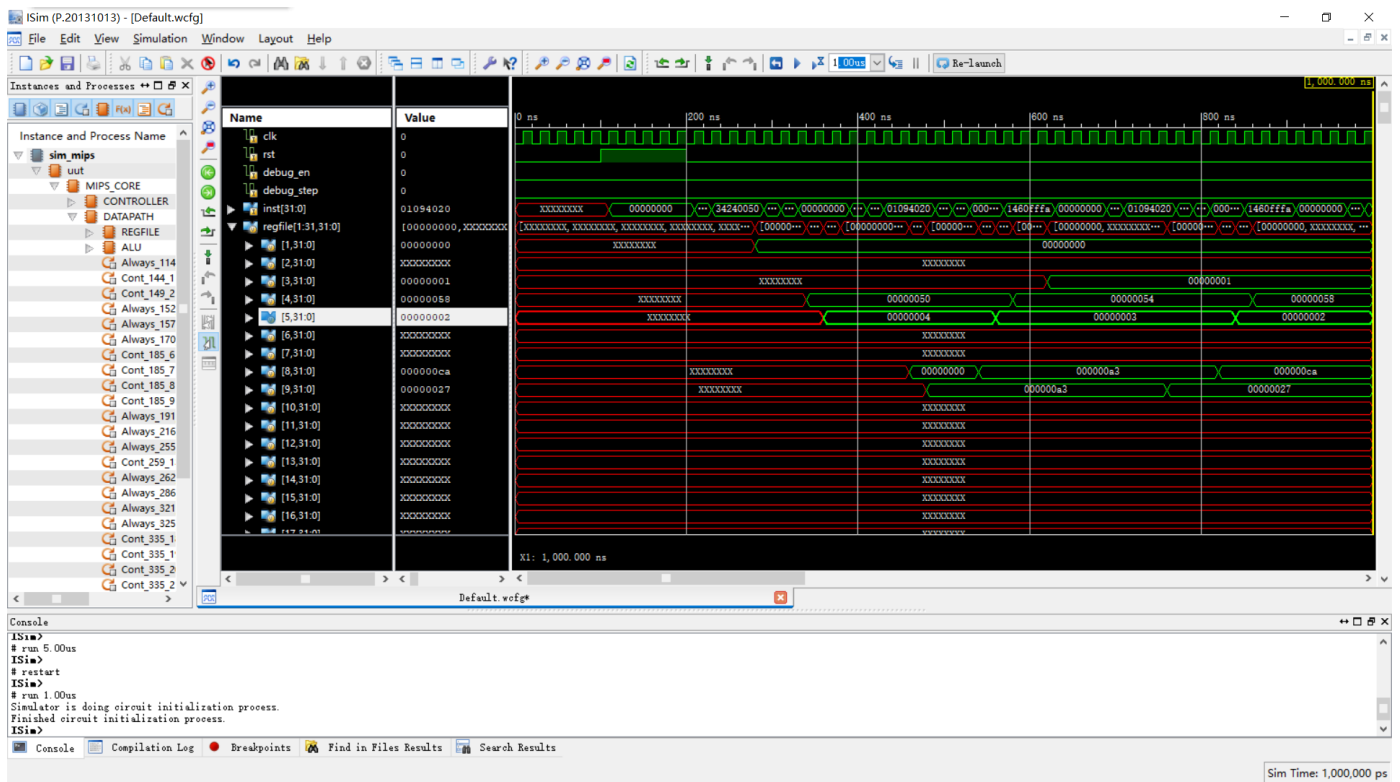
```

reg\_stall 条件达成之后，if\_en 和 id\_en 取 0，则这两个阶段我们不在执行，exe\_rst 取 1，清空上次 EXE 阶段的缓存，而 branch\_stall 条件达成之后，我们需要清空 id 阶段的缓存，也即是 id\_rst= 1;

至此完成 controller 的填空，也完成了所有工程的填空。

#### 四、实验结果分析

仿真结果：



（下文是对于仿真结果的分析，因为我们使用的测试代码和预期的实验结果和上一次实验一模一样，所以我们如果阅读过实验 1 的报告，建议跳过本节）

我们在图片右侧的紫框中找到相应的芯片元件，拖入左侧的图中就可以得到输入输出的信号波形图，注意设定仿真时长，并且显示方式为 16 进制(便于阅读)。

主要观察的波形是 regfile 和 inst 信号的输出，通过观察 inst 信号我们可以确定跳转的指令是否正确，通过观察 regfile 我们可以确定写回的信号是否正确。

我们测试用的程序如下：

```

00000824 main:      and $1, $0, $0      # address of data[0]
34240050           ori $4, $1, 80      # address of data[0]
20050004 call:      addi $5, $0, 4      # counter
0c00000a           jal sum            # call function
ac820000 return:    sw $2, 0($4)        # store result
8c890000           lw $9, 0($4)        # check sw
ac890004           sw $9, 4($4)        # store result again
01244022           sub $8, $9, $4      # sub: $8 <= $9 - $4
08000008 finish:   j finish           # dead loop
00000000           nop                # done
  
```

```

00004020 sum:      add $8, $0, $0      # sum function entry
8c890000 loop:    lw $9, 0($4)        # load data
01094020          add $8, $8, $9      # sum
20a5ffff          addi $5, $5, -1     # counter - 1
20840004          addi $4, $4, 4      # address + 4
0005182a          slt $3, $0, $5      # finish?
1460ffff          bne $3, $0, loop    # finish?
01001025          or $2, $8, $0       # move result to $v0
03e00008          jr $ra              # return
00000000          nop                # done

```

最右侧的 16 进制数是机器码，直接存放在 inst\_rom.hex 中，被 CPU 读取。左侧的内容是机器码翻译之后的汇编语言，以及汇编语言的语义（用注释表示）

设定 1 号和 4 号寄存器的数据之后，我们调用函数 sum 将 80 号寄存器的值写入 9 号寄存器，注意我们同样根据 MIPS 规则将每个寄存器存入 32 位（4 字节）的值，于是 80 号相当于第 21 个字（因为 0 号是第一个字），查找 data\_ram.hex 文件可以找到他的初始值是 a3，之后 sum 函数给 5 号寄存器不断减 1（5 号初值为 4），同时 4 号寄存器不断加 4（这样每次 9 号寄存器读取出来的值都是内存中下一个字的值）。8 号寄存器则不断累加 9 号寄存器的值，在 5 号小于零前不断循环，最终。将这个值返回给 2 号寄存器。此时四号寄存器已经是 0x60(16 进制的 60 就是十进制的 96， $= 80 + 4 * 4$ )，我们将累加的结果从 2 号寄存器存入内存中的第 25 个字，之后重新读出为 9 号寄存器的值，其值为 0x258，然后我们再减去 4 号寄存器的值 0x60,得到的结果就是八号寄存器的值，应当为 0x1f8。

于是，假若 8 号寄存器的最终值为 0x1f8，4 号为 0x60,9 号为 0x258，我们就有很大把握这个 CPU 是正确的。

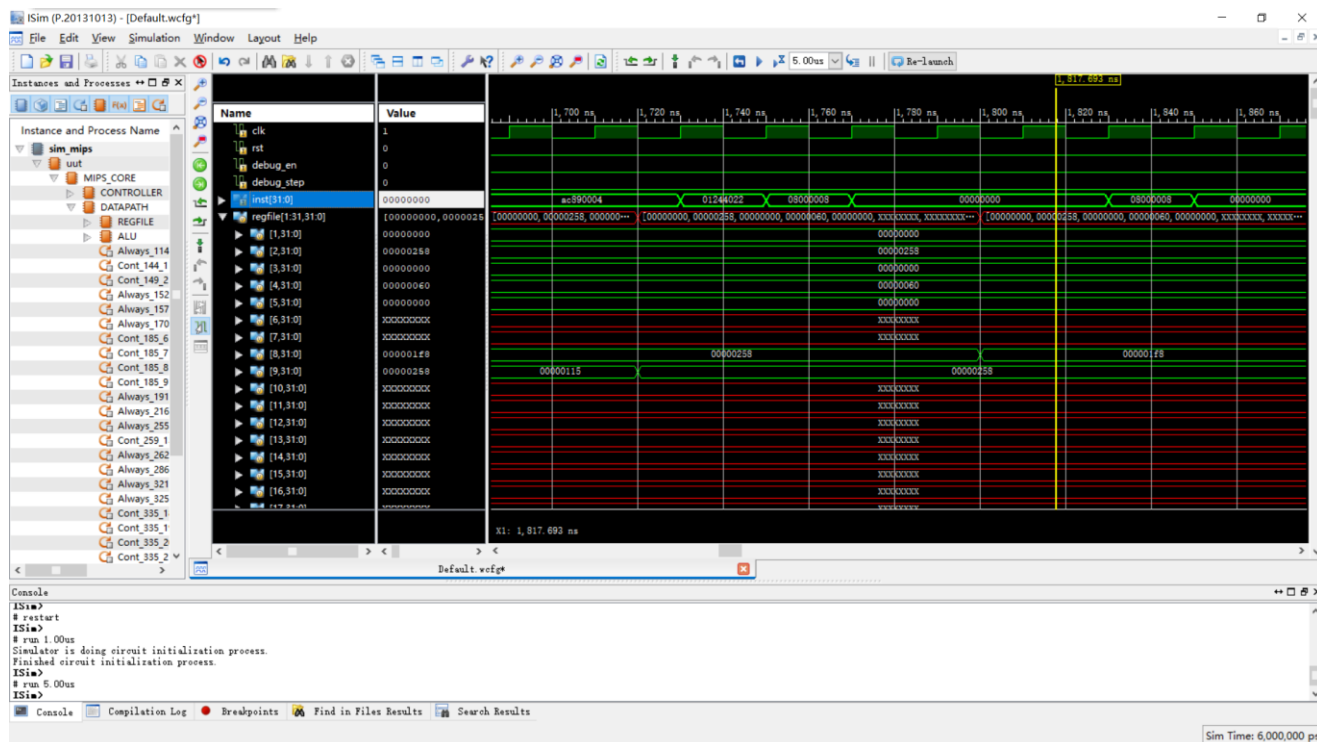
我们还可以通过 inst 寄存器的输出判断代码的执行顺序，根据汇编语言的逻辑，我们首先从 00000824 顺序执行到 0c00000a（jal 指令），之后跳转到 00004020(sum 位置) 执行到 1460ffa(bne),之后跳转为 8c890000(loop 位置),反复执行 4 次,第 4 次到达 1460ffa 之后从 01001025 向下，在 03e00008（jr 指令）出跳转回主程序 ac820000(retrun 位置) 最后顺序执行到底，在 08000008（finish 位置）重复跳转到自身，执行死循环。

Inst 部分的波形（初始）：





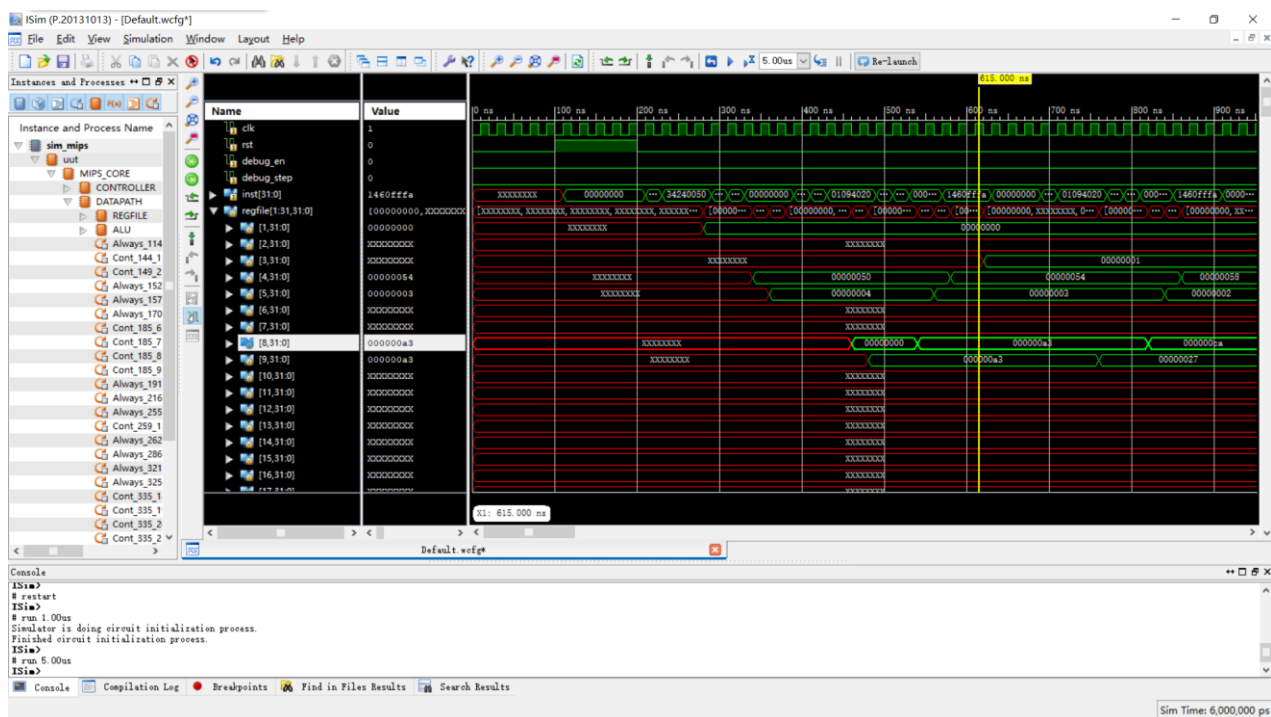
(结尾部分)



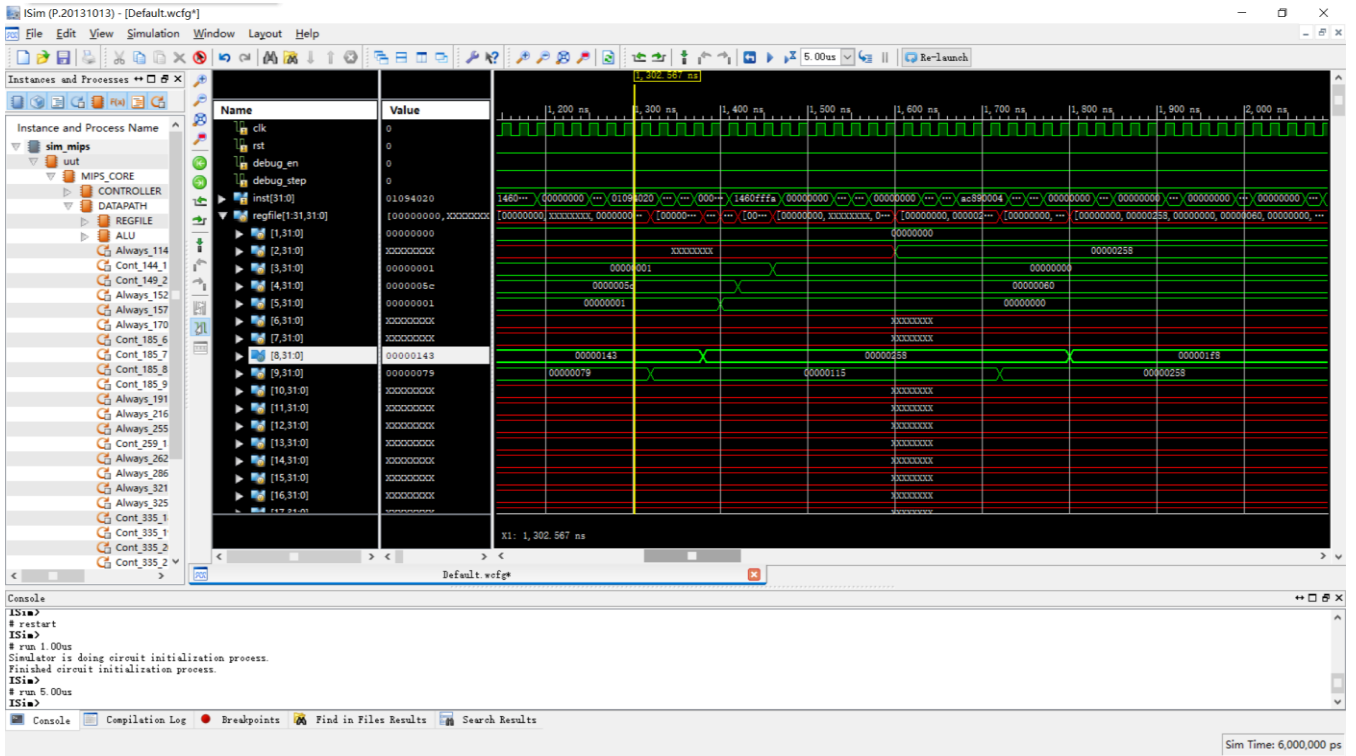
我们在进入死循环之后的 00000000 指令是因为解析死循环的 jump 指令而插入的

停顿指令。直到五阶段 `jump` 执行完毕之后继续显示 `08000008`

regfile 波形（开始部分）



结尾:



在 sword 开发板上的仿真结果（最后寄存器上的值似乎有硬件问题）



下文节选一些中间过程（但是因为首先已经跑通到了终点，所以寄存器最后的结果



一直在)

REGS-00	00000000	REGS-01	00000000	REGS-02	0000001D	REGS-03	00000000
REGS-04	00000050	REGS-05	00000004	REGS-06	00000000	REGS-07	00000000
REGS-08	000007BD	REGS-09	0000001D	REGS-0A	00000000	REGS-0B	00000000
REGS-0C	00000000	REGS-0D	0000001C	REGS-0E	00000000	REGS-0F	00000000
REGS-10	00000000	REGS-11	00000000	REGS-12	00000000	REGS-13	00000000
REGS-14	00000000	REGS-15	00000000	REGS-16	00000000	REGS-17	00000000
REGS-18	00000000	REGS-19	00000000	REGS-1A	00000000	REGS-1B	00000000
REGS-1C	00000000	REGS-1D	00000000	REGS-1E	00000000	REGS-1F	00000010
IF-ADDR	00000030	IF-INST	01094020	ID-ADDR	0000002C	ID-INST	0C890000
EX-ADDR	00000028	EX-INST	00004020	IM-ADDR	00000000	IM-INST	00000000
RS-ADDR	00000004	RS-DATA	00000050	RT-ADDR	00000003	RT-DATA	0000001D
IMMEDIAT	00000000	ALU-AIN	00000000	ALU-BIN	00000000	ALU-OUT	00000000
-----	00000000	FORWARD	00000000	MEMOPEP	00001000	MEMADDR	00000000
MEMDATR	0000063E	MEMDATW	00000000	WB-ADDR	00000000	WB-DATA	00000000
RESERVE	FFFFFFFF	RESERVE	FFFFFFFF	RESERVE	FFFFFFFF	RESERVE	FFFFFFFF
RESERVE	FFFFFFFF	RESERVE	FFFFFFFF	RESERVE	FFFFFFFF	RESERVE	FFFFFFFF
CP0S-00	00000000	CP0S-01	00000000	CP0S-02	0000001D	CP0S-03	00000000
CP0S-04	00000050	CP0S-05	00000004	CP0S-06	00000000	CP0S-07	00000000
CP0S-08	000007BD	CP0S-09	0000001D	CP0S-0A	00000000	CP0S-0B	00000000
CP0S-0C	00000000	CP0S-0D	0000001C	CP0S-0E	00000000	CP0S-0F	00000000
CP0S-10	00000000	CP0S-11	00000000	CP0S-12	00000000	CP0S-13	00000000
CP0S-14	00000000	CP0S-15	00000000	CP0S-16	00000000	CP0S-17	00000000
CP0S-18	00000000	CP0S-19	00000000	CP0S-1A	00000000	CP0S-1B	00000000
CP0S-1C	00000000	CP0S-1D	00000000	CP0S-1E	00000000	CP0S-1F	00000010
RESERVE	00000030	RESERVE	01094020	RESERVE	0000002C	RESERVE	0C890000
RESERVE	00000028	RESERVE	00004020	RESERVE	00000000	RESERVE	00000000

REGS-00	00000000	REGS-01	00000000	REGS-02	0000001D	REGS-03	00000000
REGS-04	00000050	REGS-05	00000004	REGS-06	00000000	REGS-07	00000000
REGS-08	00000000	REGS-09	0000001D	REGS-0A	00000000	REGS-0B	00000000
REGS-0C	00000000	REGS-10	0000001C	REGS-0E	00000000	REGS-0F	00000000
REGS-14	00000000	REGS-11	00000000	REGS-12	00000000	REGS-13	00000000
REGS-18	00000000	REGS-15	00000000	REGS-16	00000000	REGS-17	00000000
REGS-1C	00000000	REGS-19	00000000	REGS-1A	00000000	REGS-1B	00000000
IF-ADDR	00000034	REGS-1D	00000000	REGS-1E	00000000	REGS-1F	00000010
EX-ADDR	00000000	IF-INST	20A5FFFF	ID-ADDR	00000030	ID-INST	01094020
RS-ADDR	00000000	EX-INST	00000000	IM-ADDR	0000002C	IM-INST	0C890000
IMMEDIAT	00004020	RS-DATA	00000000	RT-ADDR	00000003	RT-DATA	0000001D
-----	00000000	ALU-AIN	00000000	ALU-BIN	00000000	ALU-OUT	00000000
MEMDATR	00000258	FORWARD	00000000	MEMOPEP	00001010	MEMADDR	00000050
RESERVE	FFFFFFFF	MEMDATW	0000001D	WB-ADDR	00000000	WB-DATA	00000000
RESERVE	FFFFFFFF	RESERVE	FFFFFFFF	RESERVE	FFFFFFFF	RESERVE	FFFFFFFF
RESERVE	FFFFFFFF	RESERVE	FFFFFFFF	RESERVE	FFFFFFFF	RESERVE	FFFFFFFF
CP0S-00	00000000	CP0S-01	00000000	CP0S-02	0000001D	CP0S-03	00000000
CP0S-04	00000050	CP0S-05	00000004	CP0S-06	00000000	CP0S-07	00000000
CP0S-08	00000000	CP0S-09	0000001D	CP0S-0A	00000000	CP0S-0B	00000000
CP0S-0C	00000000	CP0S-0D	0000001C	CP0S-0E	00000000	CP0S-0F	00000000
CP0S-10	00000000	CP0S-11	00000000	CP0S-12	00000000	CP0S-13	00000000
CP0S-14	00000000	CP0S-15	00000000	CP0S-16	00000000	CP0S-17	00000000
CP0S-18	00000000	CP0S-19	00000000	CP0S-1A	00000000	CP0S-1B	00000000
CP0S-1C	00000000	CP0S-1D	00000000	CP0S-1E	00000000	CP0S-1F	00000010
RESERVE	00000034	RESERVE	20A5FFFF	RESERVE	00000030	RESERVE	01094020
RESERVE	00000000	RESERVE	00000000	RESERVE	0000002C	RESERVE	0C890000





## 五、 讨论与心得

对于命名的规范还是需要注意的。